CIS 624: Structure of Programming Languages

Lecture 11 — STLC Extensions and Related Topics

Boyana Norris
2014

Adding Stuff

Time to use STLC as a foundation for understanding other common language constructs

We will add things via a *principled methodology* thanks to a proper education

- Extend the syntax
- Extend the operational semantics
  - Derived forms (syntactic sugar), or
  - Direct semantics
- Extend the type system
- Extend soundness proof (new stuck states, proof cases)

In fact, extensions that add new types have even more structure

Let bindings (CBV)

\[
\begin{align*}
\text{e ::= ...} & \quad \text{let } x = e_1 \text{ in } e_2 \\
& \quad e_1 \to e'_1 \\
\begin{array}{c}
\text{let } x = e_1 \text{ in } e_2 \\
\to \\
\text{let } x = e'_1 \text{ in } e_2 \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e_1 : \tau' \\
\Gamma & \vdash x : \tau' \vdash e_2 : \tau \\
\end{align*}
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau.
\]

(Also need to extend definition of substitution...)

Progress: If \( e \) is a let, 1 of the 2 new rules apply (using induction)

Preservation: Uses Substitution Lemma

Substitution Lemma: Uses Weakening and Exchange

Derived forms

\[
\begin{align*}
\text{let } x = e_1 \text{ in } e_2 & \to (\lambda x. e_2) e_1 \\
\end{align*}
\]

These 3 semantics are *different* in the state-sequence sense

\( e_1 \to e_2 \to \ldots \to e_n \)

But (totally) *equivalent* and you could prove it (not hard)

Note: ML type-checks let and \( \lambda \) differently (later topic)

Note: Don’t desugar early if it hurts error messages!

Booleans and Conditionals

\[
\begin{align*}
\text{e ::= ...} & \quad \text{true} | \text{false} | \text{if } e_1 e_2 e_3 \\
\text{v ::= ...} & \quad \text{true} | \text{false} \\
\text{τ ::= ...} & \quad \text{bool} \\
& \quad e_1 \to e'_1 \\
& \quad \text{if } e_1 e_2 e_3 \to \text{if } e'_1 e_2 e_3 \\
& \quad \text{if true } e_2 e_3 \to e_2 \\
& \quad \text{if false } e_2 e_3 \to e_3 \\
\end{align*}
\]

Also extend definition of substitution (will stop writing that)... Notes: CBN, new Canonical Forms case, all lemma cases easy
Pairs (CBV, left-right)

\[
\begin{array}{ll}
e & ::= \ldots | (e, e) | e.1 | e.2 \\
v & ::= \ldots | (v, v) \\
\tau & ::= \ldots | \tau \ast \tau \\
\end{array}
\]

\[
\begin{array}{ll}
e_1 \rightarrow e_1' & \quad e_2 \rightarrow e_2' \\
(e_1, e_2) \rightarrow (e_1', e_2') & \\
\end{array}
\]

Small-step can be a pain

- Large-step needs only 3 rules
- Will learn more concise notation later (evaluation contexts)

Records continued

Records are like n-ary tuples except with named fields

- Field names are not variables; they do not α-converge

\[
\begin{array}{ll}
e & ::= \ldots | \{l_1 = e_1; \ldots; l_n = e_n\} | e.1 \\
v & ::= \ldots | \{l_1 = v_1; \ldots; l_n = v_n\} \\
\tau & ::= \ldots | \{l_1 : \tau_1; \ldots; l_n : \tau_n\} \\
\end{array}
\]

\[
\begin{array}{ll}
e_i \rightarrow e_i' & \\
\{l_1 = v_{i-1}; \ldots; l_{i-1} = v_{i-1}; l_i = e_i'; \ldots; l_n = e_n\} & \rightarrow \{l_1 = v_1; \ldots; l_{i-1} = v_{i-1}; l_i = e_i'; \ldots; l_n = e_n\} \\
\end{array}
\]

\[
\begin{array}{ll}
1 \leq i \leq n & \\
\end{array}
\]

\[
\begin{array}{ll}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2 \\
\end{array}
\]

Canonical Forms: If \( \vdash v : \tau_1 \ast \tau_2 \), then \( v \) has the form \((v_1, v_2)\)

Progress: New cases using Canonical Forms are \(v.1\) and \(v.2\)

Preservation: For primitive reductions, inversion gives the result directly

Records continued

Should we be allowed to reorder fields?

- \( \vdash \{l_1 = 42; l_2 = \text{true}\} : \{l_2 : \text{bool}; l_1 : \text{int}\} \) ??
- Really a question about, "when are two types equal?"

Nothing wrong with this from a type-safety perspective, yet many languages disallow it

- Reasons: Implementation efficiency, type inference

Return to this topic when we study subtyping

Sums syntax and overview

\[
\begin{array}{ll}
e & ::= \ldots \mid A(e) \mid B(e) \mid \text{match } e \text{ with } A x. e \mid B x. e \\
v & ::= \ldots \mid A(v) \mid B(v) \\
\tau & ::= \ldots \mid \tau_1 + \tau_2 \\
\end{array}
\]

- Only two constructors: \( A \) and \( B \)
- All values of any sum type built from these constructors
- So \( A(e) \) can have any sum type allowed by \( e \)'s type
- No need to declare sum types in advance
- Like functions, will "guess the type" in our rules

For now, just model (1) with (anonymous) sum types

- (2) is in a later lecture, (3) is straightforward, and (4) we'll discuss informally

What about ML-style datatypes:

\[\text{type } t = A \mid B \text{ of } \text{int} \mid C \text{ of } \text{int} \ast t\]

1. Tagged variants (i.e., discriminated unions)
2. Recursive types
3. Type constructors (e.g., type \( \text{\'a mylist} = \ldots \))
4. Named types
Sums operational semantics

\[
\text{match } A(v) \text{ with } A.x. e_1 | B.y. e_2 \rightarrow e_1[v/x]
\]
\[
\text{match } B(v) \text{ with } A.x. e_1 | B.y. e_2 \rightarrow e_2[v/y]
\]
\[
e \rightarrow e' \\
A(e) \rightarrow A(e') \\
B(e) \rightarrow B(e')
\]
\[
\text{match } e \text{ with } A.x. e_1 | B.y. e_2 \rightarrow e' \text{ with } A.x. e_1 | B.y. e_2
\]

match has binding occurrences, just like pattern-matching
(Definition of substitution must avoid capture, just like functions)

What is going on

Feel free to think about tagged values in your head:

▶ A tagged value is a pair of:
  ▶ A tag A or B (or 0 or 1 if you prefer)
  ▶ The (underlying) value

▶ A match:
  ▶ Checks the tag
  ▶ Binds the variable to the (underlying) value

This much is just like OCaml and related to homework 2

Sums Typing Rules

Inference version (not trivial to infer; can require annotations)

\[
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash A(e) : \tau_1 + \tau_2 \\
\Gamma \vdash B(e) : \tau_1 + \tau_2
\]

\[
\Gamma \vdash e : \tau_1 + \tau_2 \\
\Gamma, x: \tau_1 \vdash e_1 : \tau \\
\Gamma, y: \tau_2 \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{match } e \text{ with } A.x. e_1 | B.y. e_2 : \tau
\]

Key ideas:
▶ For constructor-uses, “other side can be anything”
▶ For match, both sides need same type
  ▶ Don’t know which branch will be taken, just like an if.
  ▶ In fact, can drop explicit booleans and encode with sums:
  E.g., bool = int + int, true = A(0), false = B(0)

Sums Type Safety

Canonical Forms: If \(\cdot \vdash v : \tau_1 + \tau_2\), then there exists a \(v_1\) such
that either \(v = A(v_1)\) and \(\cdot \vdash v_1 : \tau_1\) or \(v = B(v_1)\) and
\(\cdot \vdash v_1 : \tau_2\)

▶ Progress for match \(\vdash A.x. e_1 | B.y. e_2\) follows, as usual, from Canonical Forms

▶ Preservation for match \(\vdash A.x. e_1 | B.y. e_2\) follows from the type of the underlying value and the Substitution Lemma

▶ The Substitution Lemma has new “hard” cases because we have new binding occurrences

▶ But that’s all there is to it (plus lots of induction)

Sums in C

\[
type t = A \text{ of } t_1 | B \text{ of } t_2 | C \text{ of } t_3
\]

match e with A x -> ...

One way in C:

\[
\text{struct } t \{
\text{enum } \{A, B, C\} \quad \text{tag};
\text{union } \{t_1 a; t_2 b; t_3 c;\} \text{ data;}
\};
\]

... switch(e->tag){ case A: t_1 x=e->data.a; ...}

▶ No static checking that tag is obeyed
▶ As fat as the fattest variant (avoidable with casts)
▶ Mutation costs us again!

What are sums for?

▶ Pairs, structs, records, aggregates are fundamental data-builders

▶ Sums are just as fundamental: “this or that not both”

▶ You have seen how OCaml does sums (datatypes)

▶ Worth showing how C and Java do the same thing
  ▶ A primitive in one language is an idiom in another

Sums in C

\[
type t = A \text{ of } t_1 | B \text{ of } t_2 | C \text{ of } t_3
\]

match e with A x -> ...

One way in C:

\[
\text{struct } t \{
\text{enum } \{A, B, C\} \quad \text{tag};
\text{union } \{t_1 a; t_2 b; t_3 c;\} \text{ data;}
\};
\]

... switch(e->tag){ case A: t_1 x=e->data.a; ...}

▶ No static checking that tag is obeyed
▶ As fat as the fattest variant (avoidable with casts)
▶ Mutation costs us again!
Sums in Java

```java
type t = A of t1 | B of t2 | C of t3
match e with A x -> ...
```

One way in Java (t4 is the match-expression’s type):

```java
abstract class t {abstract t m();}
class A extends t { t1 x; t4 m(){}...}
class B extends t { t2 x; t4 m(){}...}
class C extends t { t3 x; t4 m(){}...}
... e.m() ...
```

- A new method in t and subclasses for each match expression
- Supports extensibility via new variants (subclasses) instead of extensibility via new operations (match expressions)

Pairs vs. Sums

- You need both in your language
  - With only pairs, you clumsily use dummy values, waste space, and rely on unchecked tagging conventions
  - Example: replace `int + (int -> int)` with `int * (int * (int -> int))`
- Pairs and sums are "logical duals" (more on that later)
  - To make a \( \tau_1 \times \tau_2 \) you need a \( \tau_1 \) and a \( \tau_2 \)
  - To make a \( \tau_1 + \tau_2 \) you need a \( \tau_1 \) or a \( \tau_2 \)
  - Given a \( \tau_1 \times \tau_2 \), you can get a \( \tau_1 \) or \( \tau_2 \) (or both; your "choice")
  - Given a \( \tau_1 + \tau_2 \), you must be prepared for either a \( \tau_1 \) or \( \tau_2 \) (the value’s "choice")

Base Types and Primitives, in general

What about floats, strings, ...?

Could add them all or do something more general...

Parameterize our language/semantics by a collection of base types \((b_1, ..., b_n)\) and primitives \((p_1 : \tau_1, ..., p_n : \tau_n)\).

- `concat : string -> string` (the value’s "choice")
- `tolist : float -> int`
- "hello" : string

For each primitive, assume if applied to values of the right types it produces a value of the right type

Together the types and assumed steps tell us how to type-check and evaluate \( p_i v_1 ... v_n \) where \( p_i \) is a primitive

We can prove soundness once and for all given the assumptions

Recursion

We won’t prove it, but every extension so far preserves termination

A Turing-complete language needs some sort of loop, but our lambda-calculus encoding won’t type-check, nor will any encoding of equal expressive power

- So instead add an explicit construct for recursion
- You might be thinking `let rec f x = e, but we will do something more concise and general but less intuitive`

To make a \( \tau_1 \times \tau_2 \) you need a \( \tau_1 \) and a \( \tau_2 \)

Given a \( \tau_1 \times \tau_2 \), you can get a \( \tau_1 \) or \( \tau_2 \) (or both; your "choice")

Given a \( \tau_1 + \tau_2 \), you must be prepared for either a \( \tau_1 \) or \( \tau_2 \) (the value’s "choice")

Using fix

To use `fix` like `let rec`, just pass it a two-argument function where the first argument is for recursion

- Not shown: `fix` and tuples can also encode mutual recursion

Example:

```
(fix λf. λn. if (n<1) 1 (n * (f(n-1)))) 5
```

```
→ (λn. if (n<1) 1 (n * (fix λf. λn. if (n<1) 1 (n * (f(n-1))))(n-1))) 5
```

```
→ if (5<1) 1 (5 * (fix λf. λn. if (n<1) 1 (n * (f(n-1))))(5-1))
```

```
→ 5 * (fix λf. λn. if (n<1) 1 (n * (f(n-1))))(5-1) + 1
```

```
→ 5 * (fix λf. λn. if (n<1) 1 (n * (f(n-1))))(n-1)) 4
```

```
→ ...
```

Why called fix?

In math, a fix-point of a function \( g \) is an \( x \) such that \( g(x) = x \)

- This makes sense only if \( g \) has type \( \tau \rightarrow \tau \) for some \( \tau \)
- A particular \( g \) could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type \( int \rightarrow int \):

```
λx. x + 1 has no fix-points
```

```
λx. x * 0 has one fix-point
```

```
λx. absolute_value(x) has an infinite number of fix-points
```

```
λx. if (x < 10 & & x > 0) x 0 has 10 fix-points
```
Higher types

At higher types like \((\text{int} \to \text{int}) \to (\text{int} \to \text{int})\), the notion of fix-point is exactly the same (but harder to think about)

- For what inputs \(f\) of type \(\text{int} \to \text{int}\) is \(g(f) = f\)

Examples:

- \(\lambda f. \lambda x. (f \ x) + 1\) has no fix-points
- \(\lambda f. \lambda x. (f \ x) \ast 0\) (or just \(\lambda f. \lambda x. 0\)) has 1 fix-point
  - The function that always returns 0
  - In math, there is exactly one such function (cf. equivalence)
- \(\lambda f. \lambda x. \text{absolute_value}(f \ x)\) has an infinite number of fix-points: Any function that never returns a negative result

Typing \(\text{fix}\)

\[
\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \text{fix } e : \tau}
\]

Math explanation: If \(e\) is a function from \(\tau\) to \(\tau\), then \(\text{fix } e\), the fixed-point of \(e\), is some \(\tau\) with the fixed-point property

- So it's something with type \(\tau\)

Operational explanation: \(\text{fix } \lambda x. e'\) becomes \(e'[\text{fix } \lambda x. e'/x]\)

- The substitution means \(x\) and \(\text{fix } \lambda x. e'\) need the same type
- The result means \(e'\) and \(\text{fix } \lambda x. e'\) need the same type

Note: The \(\tau\) in the typing rule is usually insantiated with a function type

- e.g., \(\tau_1 \to \tau_2\), so \(e\) has type \((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)\)

Note: Proving soundness is straightforward!

Anonymous

We added many forms of types, all \textit{unnamed} a.k.a. \textit{structural}.
Many real PLs have (all or mostly) \textit{named} types:

- Java, C, C++: all record types (or similar) have names
  - Omitting them just means compiler makes up a name
- OCaml sum types and record types have names

A never-ending debate:

- Structural types allow more code reuse: good
- Named types allow less code reuse: good
- Structural types allow generic type-based code: good
- Named types let type-based code distinguish names: good

The theory is often easier and simpler with structural types

Back to factorial

Now, what are the fix-points of \(\lambda f. \lambda x. \text{if } (x < 1) 1 (x \ast (f(x-1)))\)?

It turns out there is exactly one (in math): the factorial function!

And \(\text{fix } \lambda f. \lambda x. \text{if } (x < 1) 1 (x \ast (f(x-1)))\) behaves just like the factorial function

- That is, it behaves just like the fix-point of \(\lambda f. \lambda x. \text{if } (x < 1) 1 (x \ast (f(x-1)))\)
- In general, \(\text{fix}\) takes a function-taking-function and returns its fix-point

(This isn’t necessarily important, but it explains the terminology and shows that programming is deeply connected to mathematics)

Termination

Surprising fact: If \(\vdash e : \tau\) in STLC with all our additions \textit{except} \textit{fix}, then there exists a \(v\) such that \(e \rightarrow^* v\)

- That is, all programs terminate

So termination is trivially decidable (the constant “yes” function), so our language is not Turing-complete

The proof requires more advanced techniques than we have learned so far because the size of expressions and typing derivations does not decrease with each program step

Non-proof:

- Recursion in \(\lambda\) calculus requires some sort of self-application
- Easy fact: For all \(\Gamma, x,\) and \(\tau\), we \textit{cannot} derive \(\Gamma \vdash x : \tau\)